Automatic Parallelization via Matrix Multiplication

Shigeyuki Sato Hideya Iwasaki
The University of Electro-Communications, Tokyo, Japan
sato@ipl.cs.uec.ac.jp iwasaki@cs.uec.ac.jp

Abstract

Existing work that deals with parallelization of complicated reductions and scans focuses only on formalism and hardly dealt with implementation. To bridge the gap between formalism and implementation, we have integrated parallelization via matrix multiplication into compiler construction. Our framework can deal with complicated loops that existing techniques in compilers cannot parallelize. Moreover, we have sophisticated our framework by developing two sets of techniques. One enhances its capability for parallelization by extracting max-operators automatically, and the other improves the performance of parallelized programs by eliminating redundancy. We have also implemented our framework and techniques as a parallelizer in a compiler. Experiments on examples that existing compilers cannot parallelize have demonstrated the scalability of programs parallelized by our implementation.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Compilers, Optimization; D.1.2 [*Programming Techniques*]: Automatic Programming

General Terms Experimentation, Languages, Design, Algorithms

Keywords automatic parallelization, loop, reduction, scan, matrix multiplication, semiring, linear recurrence equation

1. Introduction

Since 2005, processor vendors have generally adopted multi-core architectures instead of boosting the clock rate of processors. This means that sequential programs cannot be made run faster without parallelization. Thus, we cannot avoid parallel programming in striving for higher performance. However, parallel programming is a challenge for most programmers. The easiest way for programmers to make programs parallel is to use automatic parallelization.

The most commonly used methodology for automatic parallelization of loops is *doall parallelization* [1, 2], whose core is to guarantee the independence of each iteration, i.e., the parallelism among iterations, by analyzing *loop-carried data dependence*. This framework suffices for simple data parallelism, but does not suffice for *reduction*, which is a generalization of summation.

Standard doall parallelizers can recognize simple reductions, e.g., one that just computes the sum of an array:

$$x \leftarrow 0;\ \textbf{for}\ i = 1\ \textbf{to}\ n\ \textbf{do}\ x \leftarrow x + a[i]\ \textbf{done}.$$

PLDI'11, June 4–8, 2011, San Jose, California, USA.

This loop is equivalent to $x \leftarrow 0+\sum_{i=1}^{n} a[i]$. As is well known, the summation can be computed in parallel owing to the *associativity* of the addition; the parallel summation is computed in $\mathrm{O}(n/p + \log p)$ time, where p is the number of threads. Although this loop has a loop-carried data dependence with respect to x (i.e., writing x after reading x over an iteration), its doall parallelization succeeds because the definition and use of x are recognized as a reduction.

The summation is a trivial reduction. There are, however, more unobvious and non-trivial reductions. For example, the following loop evaluates a polynomial expression through the Horner scheme, i.e., $\sum_{i=0}^{n} a[n-i]c^i = a[n] + c(a[n-1] + c(a[n-2] + \cdots + c(a[1] + c(a[0] + c \cdot 0)) \cdots))$.

$$x \leftarrow 0;\ \textbf{for}\ i = 0\ \textbf{to}\ n\ \textbf{do}\ x \leftarrow c \cdot x + a[i]\ \textbf{done}.$$

Doall parallelizers cannot recognize this loop as a reduction. As a result, this loop-carried data dependence with respect to x makes its doall parallelization impossible. However, we can rewrite this loop into

$$\boldsymbol{x} \leftarrow \boldsymbol{x}_0;\ \textbf{for}\ i = 0\ \textbf{to}\ n\ \textbf{do}\ \boldsymbol{x} \leftarrow A_i \times \boldsymbol{x}\ \textbf{done},$$
$$\textbf{where}\ \boldsymbol{x} = \begin{pmatrix} x \\ 1 \end{pmatrix},\ \boldsymbol{x}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix},\ A_i = \begin{pmatrix} c & a[i] \\ 0 & 1 \end{pmatrix}.$$

This loop is equivalent to $\boldsymbol{x} \leftarrow \left(\prod_{i=0}^{n} A_{n-i}\right) \times \boldsymbol{x}_0$. It is the same as the summation except for changing the addition into the matrix multiplication. Since the matrix multiplication is an associative operation, we can also compute the product of matrices in $\mathrm{O}(n/p + \log p)$ time. Thus, this loop can be parallelized.

As shown above, if we can transform a loop body into a matrix multiplication form, we can obtain an efficient parallel version of its loop. This technique is known as a parallel algorithm for solving linear recurrence equations [10]. There is work that applies a generalization of this idea to automatic parallelization [6, 8, 13, 14, 21]. Although the formalism developed in such work is promising for parallelizing complicated loops, no one connects the formalism with the implementation aspect. We have resolved this problem by integrating the formalism into doall parallelization.

Our work to bridge the gap between formalism and practical implementation includes a solution to an important problem concerning the max-operator, which plays a key role in parallelizing dynamic programming. In prior work on algebra-based parallelization [13, 21], this operator is assumed to be given, even though in realistic programs the max-operation is usually described by means of if-statements. We have developed a technique for extracting max-operators automatically from if-statements.

Our work has resulted in the following important contributions.

- We have developed a novel framework for loop parallelization (Sections 3 and 4). The formalization of a loop body by using matrix multiplication over a semiring enables parallelization of various loops, especially one with a complicated body that contains loop-carried data dependence and if-statements. This parallelization is more powerful than the standard doall

parallelization. Our framework is an integration of doall parallelization and the existing formalism [6, 8, 13, 14, 21]. To the best of our knowledge, our work is the first to deal seriously with the implementation aspect of deriving associative operators for automatic parallelization.

- We have developed a technique for extracting max-operators automatically and semantically from if-statements by using an off-the-shelf Satisfiability Modulo Theories (SMT) solver (Section 5). This technique plays an important role in our framework for parallelizing various loops whose bodies contain if-statements. Automatic extraction of max-operators has not been addressed in algebra-based parallelization [13, 21]. To the best of our knowledge, our work is the first to deal seriously with the max-plus semiring in the practical context of automatic parallelization.
- We have developed optimization techniques for loops that are parallelized by our framework (Section 6). Because these techniques reduce the intrinsic overhead caused by our parallelization, they are essential in practical situations. One of these techniques is also helpful for doall parallelization (Section 6.4).
- We have implemented our framework and techniques as a parallelizer in a realistic compiler. We have then conducted experiments on examples whose doall parallelization fails and have confirmed the scalability of these versions parallelized by our implementation (Section 7).

2. Preliminaries

2.1 Notations

For program description in this paper, we use a mixture of mathematical operators and syntax constructs of standard procedural languages like Fortran and C.

The two sides of an assignment are connected by $\leftarrow$. If each side is a vector, it is a vectorized assignment, which denotes simultaneous assignment of every entry on the right side to the corresponding entry on the left side. Note that vector and matrix in this paper are virtual, i.e., not an actual data structure. They are simply syntax constructs for expressing vectorized assignment and matrix multiplication. Operators $\cdot$ and $\times$ denote scalar and matrix multiplication, respectively. We omit $\cdot$ when it is obvious. A sequence enclosed by brackets, e.g., $[a_1, \ldots, a_n]$, is an extensible notation of an array, i.e., enumerated values are stored in the array in order. Statement are separated by a semicolon.

for $i = 1$ **to** n **do** b **done** is a for-loop with a control variable of i and a body of b. Unless otherwise noted, n denotes the number of iterations. **if** e **then** b_1 **else** b_2 **endif** is an if-statement. $(e_1)\ ?\ e_2 : e_3$ is a conditional expression using the syntax of C. **parallel** $k \in \{1, \ldots, p\}$ **do** b **done** is a parallel block in which b is executed by p threads in parallel, where k denotes thread ID. Unless otherwise noted, p denotes the number of threads. Here, the shared memory model, i.e., EREW PRAM, and $p \ll n$ are assumed.

2.2 Terminologies

Techniques for identification and parallelization of the *doall loop*, which is a loop whose all iterations can be executed in parallel, have been well studied [1, 2]. In this paper, we use the term *doall parallelization* for these techniques.

To identify a loop as a doall loop, a doall parallelizer analyzes the *loop-carried data dependence*. There is a loop-carried (true) data dependence with respect to a variable (or array element) x iff x is defined in an iteration and x is used in the succeeding iterations where its definition reaches. If there is no loop-carried data dependence in a loop, the loop is identified as a doall loop.

Consider **for** $i = 1$ **to** n **do** $x \leftarrow a_i$ **done**. Because there is no loop-carried data (true) dependence, we can transform this loop into its doall version. However, in its parallelized version, the result of x is nondeterministic. In this case, at the end of the loop, x must be a_n, which is called the *final value* of x, due to its sequential semantics. A guarantee that the final values of variables defined in a parallelized loop coincide with those in its sequential version is called the *final value guarantee*.

We define several terms here.

Definition 1 (Accumulator)**.** A scalar variable x is an accumulator iff there is loop-carried data dependence with respect to x. Unless otherwise noted, x is an accumulator.

Definition 2 (Recurring expression and symbolically constant expression)**.** An expression e in a loop is a recurring expression iff e is computed from some loop-carried value. If e is not a recurring expression, we call e a symbolically constant expression. $\mathcal{C}$ denotes the set of symbolically constant expressions in a given program.

2.3 Target Loops

The target of the proposed parallelization is a non-nested loop the body of which contains neither jumps (e.g., goto, break, and continue), labels, pointers, indirect access to arrays, nor function calls; i.e., a target loop body contains only assignments, if-statements, and side-effect-free expressions. We call a sequence of assignments and if-statements a *block*.

For convenience, the start value of the control variable of each target loop is normalized to 1 using induction variable transformation [1, 2]. There are two characteristic restrictions on target loops:

- There is no loop-carried data dependence with respect to any array element.
- If an accumulator occurs in the condition-part of an if-statement, there is no assignment to the accumulator in either the then-part or else-part.

We can overcome the first restriction by using scalar replacement [5]. A technique for overcoming the second restriction is presented in Section 5, but the explanation in Sections 3 and 4 supposes this restriction. This paper does not deal with programs that cause runtime errors, and ignores arithmetic overflow and round-off errors.

2.4 Parallel Primitives

We introduce parallel primitives, which are generic patterns of parallel computation. The definitions of primitives used are:

$$\mathsf{reduce}(\odot, [x_1, \ldots, x_n]) = x_n \odot \cdots \odot x_1$$

$$\mathsf{scan}(\odot, e, [x_1, \ldots, x_n]) = [e, x_1 \odot e, \ldots, x_{n-1} \odot \cdots \odot x_1 \odot e],$$

where $\odot$ is an associative operator. Although the definitions of reduce and scan above are a bit different from these traditional definitions for convenience, it does not matter essentially.

The reduce algorithm consists of two phases: *local reduction* and *global reduction*. In the local reduction, p threads reduce n elements into p elements independently. In the global reduction, p threads reduce the result of the local reduction into the final result cooperatively. For example, consider $\mathsf{reduce}(+, [a_1, \ldots, a_n])$. In the local reduction, the k-th thread computes $r_k \leftarrow \sum_{(k-1)n/p+1}^{kn/p} a_i$. In the global reduction, p threads compute $\sum_1^n a_i$ through $\sum_1^p r_i$. The time complexity of the local reduction is $\mathrm{O}(n/p)$ and that of the global reduction is $\mathrm{O}(\log p)$. $\mathrm{O}(p)$ space is used.

The scan algorithm consists of three phases: *local reduction*, *global scan*, and *local scan*. The local reduction is the same as that of reduce. In the global scan, p threads produce new p elements from the result of the local reduction cooperatively. In the local scan, p threads compute the final result from the re-

sult of the global scan and the input sequence. For example, consider $\mathsf{scan}(+, e, [a_1, \ldots, a_n])$. In the local reduction, the k-th thread computes $r_k \leftarrow \sum_{(k-1)n/p+1}^{kn/p} a_i$. In the global scan, p threads compute $r' \leftarrow \left[e, e + \sum_1^1 r_i, \ldots, e + \sum_1^{p-1} r_i\right]$. In the local scan, the k-th thread computes $\left[e + \sum_1^{(k-1)n/p} a_i, \ldots, e + \sum_1^{kn/p-1} a_i\right]$ from $r'[k-1]$ and the subarray of a. The time complexity of the global scan is $\mathrm{O}(\log p)$ and that of the local scan is $\mathrm{O}(n/p)$. $\mathrm{O}(p)$ space is used for intermediate data.

In the shared memory model, we can simply describe each of the local reduction and local scan as a parallel loop with an iteration space that is block-partitioned. We can merge a doall loop followed by a scan or reduce into the local reduction of the reduce or the local scan of the scan if the number of iterations in the doall loop is the same as the length of an array that the scan or reduce computes. The global reduction and global scan are described in detail elsewhere [18].

3. Formalization based on Matrix Multiplication

Before describing our framework for parallelization, we present examples that are used in this section.

Example 1. Although the following loop contains a simple assignment other than a reduction, doall parallelizers can also handle it.

$$x \leftarrow 0;\ \textbf{for}\ i = 1\ \textbf{to}\ n\ \textbf{do}\ x \leftarrow x + a[i];\ a[i] \leftarrow a[i] + 1\ \textbf{done}.$$

Doall parallelizers deal with the above loop by recognizing part of a trivial reduction in an ad hoc way. Thus, they cannot parallelize non-trivial reductions that they cannot recognize as reduction. An example of non-trivial reductions is the polynomial evaluation shown in Section 1. Another example is as follows.

Example 2. The following loop computes the *maximum tail-segment sum* (mts) of a given array, i.e., the maximum sum of a contiguous segment that contains the last element of a given array; e.g., the mts of $[2, -1, -3, 5, 0, -4, 6]$ is $5 + 0 + (-4) + 6 = 7$.

$x \leftarrow 0;$
for $i = 1$ **to** n **do**
  if $x + a[i] \geq 0$ **then** $x \leftarrow x + a[i]$ **else** $x \leftarrow 0$ **endif done**.

This is a non-trivial reduction because its loop body contains an if-statement whose condition-part contains an accumulator.

In addition, doall parallelization cannot support using the intermediate results of reduction for other computations.

Example 3. The following loop computes the prefix sum of an array, i.e., $\left[\sum_{i=1}^1 a[i], \cdots, \sum_{i=1}^n a[i]\right]$. Although it resembles summation, its doall parallelization fails due to $a[i] \leftarrow x$.

$$x \leftarrow 0;\ \textbf{for}\ i = 1\ \textbf{to}\ n\ \textbf{do}\ x \leftarrow x + a[i];\ a[i] \leftarrow x\ \textbf{done}.$$

Example 4. The following loop resembles Example 2; it computes the start position of the mts of an array. For example, the start position of the mts of $[2, -1, -3, 5, 0, -4, 6]$ is 4.

$x_1 \leftarrow 0;\ x_2 \leftarrow 1;$
for $i = 1$ **to** n **do**
  if $x_1 + a[i] \geq 0$ **then** $x_1 \leftarrow x_1 + a[i]$
    else $x_1 \leftarrow 0;\ x_2 \leftarrow i + 1$ **endif done**,

where x_1 is the mts and x_2 is its start position. Note that $x_2 \leftarrow i+1$ implicitly uses the intermediate result of x_1 since it is in an if-statement whose condition-part uses x_1. Here, we assume that x_1 is not used after this loop for convenience. The start position of the mts is important for application; e.g., if $a[i]$ be the price of a stock at time i, x_2 is the time when the price of the stock began to rise.

3.1 Matrix Multiplication Form

Recall the example of polynomial evaluation shown in Section 1; its loop has a loop-carried data dependence with respect to x, which makes doall parallelization impossible. However, we can transform its loop body into the following form:

$$\boldsymbol{x} \leftarrow A_i \times \boldsymbol{x}, \quad \textbf{where}\ \boldsymbol{x} = \begin{pmatrix} x \\ 1 \end{pmatrix},\ A_i = \begin{pmatrix} c & a[i] \\ 0 & 1 \end{pmatrix}.$$

Note that the second entry of the result of $A_i \times \boldsymbol{x}$ is always 1. We permit a constant to be in the left side of an assignment if both sides are always the same. From the above transformation, the loop turns out to be equivalent to $\boldsymbol{x} \leftarrow A_n \times \cdots \times A_0 \times \boldsymbol{x}$. Owing to the associativity of $\times$, we obtain the following assignment:

$$\boldsymbol{x} \leftarrow \mathsf{reduce}\,(\times, [A_0, \ldots, A_n]) \times \boldsymbol{x}.$$

Since we can transform this loop into a reduce, we can compute it in $\mathrm{O}(n/p + \log p)$ time. Note that we can combine the generation of A_i from array element $a[i]$ with the local reduction of reduce. The key point of this parallelization is to transform the loop body into matrix-vector multiplication by introducing $\boldsymbol{x}$ and A_i. Then, by unfolding all iterations, we can obtain a chain of matrix multiplications, i.e., a reduction.

We can generalize the above idea over a semiring[1]. We use $(R, \oplus, \otimes)$ to denote a semiring, and $\mathbf{0}$ and $\mathbf{1}$ to denote identity elements regarding $\oplus$ and $\otimes$, respectively. An operator $\times_{\{\oplus,\otimes\}}$ denotes matrix multiplication over $(R, \oplus, \otimes)$, i.e., $\oplus$ and $\otimes$ of $\times_{\{\oplus,\otimes\}}$ correspond to $+$ and $\cdot$ of $\times$, respectively. $(\mathbb{R}, +, \cdot)$, $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$, and $(\{0, 1\}, \vee, \wedge)$, where $\uparrow$ is the max-operator defined as $x \uparrow y = (x < y)\ ?\ y : x$, are useful instances of semirings.

Definition 3 (Matrix multiplication form)**.** A block is in matrix multiplication form iff it is structured as follows:

$$\begin{pmatrix} x_1 \\ \vdots \\ x_m \\ \mathbf{1} \end{pmatrix} \leftarrow \begin{pmatrix} e_{11} & \cdots & e_{1m} & e_{10} \\ \vdots & \ddots & \vdots & \vdots \\ e_{m1} & \cdots & e_{1m} & e_{m0} \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{1} \end{pmatrix} \times_{\{\oplus,\otimes\}} \begin{pmatrix} x_1 \\ \vdots \\ x_m \\ \mathbf{1} \end{pmatrix},$$

$$\textbf{where}\ e_{jk} \in \mathcal{C}.$$

Theorem 1. *If a loop body is in matrix multiplication form, the loop can be computed in* $\mathrm{O}(n/p + \log p)$ *time through* reduce.

As an example, consider Example 2, which has a body of

$$\textbf{if}\ x + a[i] \geq 0\ \textbf{then}\ x \leftarrow x + a[i]\ \textbf{else}\ x \leftarrow 0\ \textbf{endif}.$$

If this if-statement is equivalent to an assignment $x \leftarrow (x + a[i]) \uparrow 0$, it can be transformed into the following matrix multiplication form over $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$.

$$\boldsymbol{x} \leftarrow A_i \times_{\{\uparrow,+\}} \boldsymbol{x}, \quad \textbf{where}\ \boldsymbol{x} = \begin{pmatrix} x \\ 0 \end{pmatrix},\ A_i = \begin{pmatrix} a[i] & 0 \\ -\infty & 0 \end{pmatrix}.$$

Thus, this loop becomes a parallel reduction: $\boldsymbol{x} \leftarrow \mathsf{reduce}(\times_{\{\uparrow,+\}}, [A_1, \ldots, A_n]) \times_{\{\uparrow,+\}} \boldsymbol{x}$. The success of this parallelization owes much to the transformation of the original if-statement into an expression by means of the max-operator. Thus, in our parallelization, it is quite important to extract the max-operator from the original loop body, which contains (possibly nested) if-statements. Section 5 shows our solution to the problem of max-operator extraction.

[1] A semiring is an algebraic structure similar to a ring, but without the requirement that each element must have an additive inverse.

Of course, we can also transform a simple reduction that doall parallelizers can recognize into a matrix multiplication form. For instance, the loop body of the summation loop in Section 1 is:

$$\boldsymbol{x} \leftarrow A_i \times_{\{+,\cdot\}} \boldsymbol{x}, \quad \textbf{where } \boldsymbol{x} = \begin{pmatrix} x \\ 1 \end{pmatrix},\ A_i = \begin{pmatrix} 1 & a[i] \\ 0 & 1 \end{pmatrix}.$$

3.2 Normal Form

Although the matrix multiplication form covers non-trivial reductions, it is not satisfactory because it cannot express either doall computation, e.g., Example 1, or use of the intermediate results of reduction for other computations, e.g., Examples 3 and 4. Therefore, we need a more general way of parallelization based on matrix multiplication to enhance its applicability in practical situations.

To understand this generalization, recall the loop of Example 1:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } x \leftarrow x + a[i];\ a[i] \leftarrow a[i] + 1 \textbf{ done}.$$

Its loop body consists of a reduction part ($x \leftarrow x + a[i]$) and an increment part ($a[i] \leftarrow a[i] + 1$). We cannot transform this body into a matrix multiplication form. However, observing that the reduction and increment parts are independent computations, we find that this example's loop can be divided into two loops:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } x \leftarrow x + a[i] \textbf{ done};$$
$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } a[i] \leftarrow a[i] + 1 \textbf{ done}.$$

Both of these two loops are parallelizable; the first is a summation loop shown in Section 1 and the second is a simple doall loop. In this way, we parallelize the loop of Example 1.

Next, recall the loop of Example 3:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } x \leftarrow x + a[i];\ a[i] \leftarrow x \textbf{ done}.$$

Again, it is impossible to transform this loop body into a matrix multiplication form. However, by computing and storing all intermediate values of x into a temporary array in advance, we can divide this loop into the following two loops:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } t[i] \leftarrow x;\ x \leftarrow x + a[i] \textbf{ done};$$
$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } a[i] \leftarrow a[i] + t[i] \textbf{ done}.$$

These two loops are parallelizable because the first is equivalent to $t \leftarrow \mathsf{scan}(+, x, a);\ x \leftarrow x + t[n]$ and the second is a doall loop. Here, we can eliminate t completely by fusing the local scan of $\mathsf{scan}(+, x, a)$ and the doall version of the second loop.

We can use the above techniques to generalize the matrix multiplication form to the following *normal form*.

Definition 4 (Normal form)**.** A block is in the normal form iff it is structured as follows:

$$\boldsymbol{t} \leftarrow A \times_{\{\oplus,\otimes\}} \boldsymbol{x};\ b;\ \boldsymbol{x} \leftarrow \boldsymbol{t},$$

where $\boldsymbol{t}$ is a vector of temporary variables, $A \times_{\{\oplus,\otimes\}} \boldsymbol{x}$ is the same as the right hand side of the matrix multiplication form, and b is a block of computations that contain no assignment to any accumulator. We call b the *auxiliary part*. Note that a matrix multiplication form can be transformed into a normal form the auxiliary part of which is empty.

The normal form is used to easily distinguish the reduce or scan computation that can be expressed by means of matrix multiplication from other computations. After obtaining a body in the normal form, we can straightforwardly parallelize a loop that has the body by dividing it into two parallel computations: 1) reduce or scan with $\times_{\{\oplus,\otimes\}}$ and 2) a doall computation for the auxiliary part. Note that the latter can be efficiently embedded into the former.

Theorem 2. *If a loop body is in the normal form, the loop can be computed in* $O(n/p + \log p)$ *time with* $O(p)$ *space for intermediate data, by using the* reduce *algorithm or the* scan *algorithm.*

3.3 Separable Normal Form

Sometimes, a single loop contains two (or more) reduction/scan computations. In such cases, although it is impossible to transform the loop body into a single normal form, we can extract a normal form from the body and thereby obtain a sequence of loops, each of which has a normal form.

For example, recall the loop body of Example 4:

$$\textbf{if } x_1 + a[i] \geq 0 \textbf{ then } x_1 \leftarrow x_1 + a[i] \textbf{ else } x_1 \leftarrow 0;\ x_2 \leftarrow i + 1 \textbf{ endif}.$$

This loop body has two reduction computations for x_1 and x_2. The first is $x_1 \leftarrow (x_1 + a[i] \geq 0)\ ?\ x_1 + a[i]\ :\ 0$. The second is $x_2 \leftarrow (x_1 + a[i] \geq 0)\ ?\ x_2 : i + 1$. We can transform the first into the following normal form:

$$\boldsymbol{t} \leftarrow A_i \times_{\{\uparrow,+\}} \boldsymbol{x};\ \boldsymbol{x} \leftarrow \boldsymbol{t},$$
$$\textbf{where } \boldsymbol{t} = \begin{pmatrix} t \\ 0 \end{pmatrix},\ A_i = \begin{pmatrix} a[i] & 0 \\ -\infty & 0 \end{pmatrix},\ \boldsymbol{x} = \begin{pmatrix} x_1 \\ 0 \end{pmatrix}.$$

Now, we expand t to an array and define $\boldsymbol{t}_i = \begin{pmatrix} t[i] & 1 \end{pmatrix}^T$. Because $t[i]$ contains the value of x_1 used in the second statement at the i-th iteration of the loop, we can transform the original loop as follows:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } \boldsymbol{t}_i \leftarrow A_i \times_{\{+,\cdot\}} \boldsymbol{x};\ \boldsymbol{x} \leftarrow \boldsymbol{t}_i \textbf{ done};$$
$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } x_2 \leftarrow (t[i] + a[i] \geq 0)\ ?\ x_2 : i + 1 \textbf{ done}.$$

Next, we can transform this second loop's body:

$$\boldsymbol{t}' \leftarrow A'_i \times_{\{+,\cdot\}} \boldsymbol{x}';\ \boldsymbol{x}' \leftarrow \boldsymbol{t}',$$
$$\textbf{where } \boldsymbol{t}' = \begin{pmatrix} t' \\ 0 \end{pmatrix},\ \boldsymbol{x}' = \begin{pmatrix} x_2 \\ 0 \end{pmatrix},\ c = (t[i] + a[i] \geq 0),$$
$$A' = \begin{pmatrix} c\ ?\ 1 : 0 & c\ ?\ 0 : i + 1 \\ 0 & 1 \end{pmatrix}.$$

We can parallelize both loops on the basis of Theorem 2. Thus, we have obtained a sequence of parallelizable loops:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } \boldsymbol{t}_i \leftarrow A_i \times_{\{\uparrow,+\}} \boldsymbol{x};\ \boldsymbol{x} \leftarrow \boldsymbol{t}_i \textbf{ done};$$
$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } \boldsymbol{t}' \leftarrow A'_i \times_{\{+,\cdot\}} \boldsymbol{x}';\ \boldsymbol{x}' \leftarrow \boldsymbol{t}' \textbf{ done}.$$

In this case, the second loop encodes the final value guarantee for x_2. We can implement the final value guarantee more efficiently (see Section 6.4).

Definition 5 (Separable normal form)**.** Let $b_1;\ b_2$ be a sequence of two blocks. b_1 is a separable normal form iff b_1 is in the normal form and no accumulator updated in b_1 is updated in b_2. In addition, if b_2 is also a separable normal form, we call $b_1;\ b_2$ a sequence of separable normal forms.

Corollary 1. *If a loop body is a sequence of separable normal forms, the loop can be computed in* $O(n/p + \log p)$ *time.*

4. Parallelization Algorithm

As shown in Section 3, once we have obtained a normal form (or a sequence of separable normal forms) of a given loop body, we can parallelize the loop straightforwardly in a divide-and-conquer manner: reduce or scan. Therefore, the core of parallelization algorithms of our framework is to extract a normal form from a given loop body. In this section, we describe algorithms for extracting it.

4.1 Extracting a Normal Form

We describe an algorithm for transforming a loop body into a normal form. It consists of two phases: separation and extraction.

In the separation phase, a given block b is split into blocks b_1 and b_2 such that b_1 contains only the updating of accumulators

found in b, and b_2 corresponds to the residual computation. We call b_1 the *reduction part* and b_2 the *residual part*. For example, consider the following loop body:

$$\textbf{if } a[i] < 0 \textbf{ then } x \leftarrow x + a[i] \textbf{ else } x \leftarrow 2x \textbf{ endif } a[i] \leftarrow x + 1.$$

We can split this loop body into a reduction part,

$$\textbf{if } a[i] < 0 \textbf{ then } x \leftarrow x + a[i] \textbf{ else } x \leftarrow 2x \textbf{ endif},$$

and a residual part,

$$\textbf{if } a[i] < 0 \textbf{ then } t_x \leftarrow x + a[i] \textbf{ else } t_x \leftarrow 2x \textbf{ endif}; \ a[i] \leftarrow t_x + 1,$$

where t_x is a temporary variable. The residual part directly corresponds to the auxiliary part in the normal form.

In the extraction phase, a matrix multiplication form is extracted from the reduction part. First, all if-statements are converted into assignments with conditional expressions, by inserting self-assignments of accumulators as needed. For example,

$$\textbf{if } a[i] < 0 \textbf{ then } x \leftarrow x + a[i] \textbf{ else } \textbf{ endif},$$

can be converted into

$$x \leftarrow (a[i] < 0) \ ? \ x + a[i] : x.$$

Next, the reduction part, which is a sequence of assignments, is converted into a single vectorized assignment by using symbolic substitution of definitions of accumulators to uses of accumulators as needed. For example,

$$x_1 \leftarrow x_1 + x_2 + a[i]; \ x_2 \leftarrow a[i] \cdot x_1 + x_2,$$

can be converted into

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + x_2 + a[i] \\ a[i] \cdot (x_1 + x_2 + a[i]) + x_2 \end{pmatrix}.$$

Finally, each entry on the right side of the vectorized assignment is transformed into a matrix multiplication form, by expanding and simplifying each expression on the basis of the axioms of a semiring. For example, the vectorized assignment above can be transformed into

$$\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 & a[i] \\ a[i] & a[i]+1 & a[i] \cdot a[i] \\ 0 & 0 & 1 \end{pmatrix} \times_{\{+,\cdot\}} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}.$$

We intuitively and informally sketches how to extract a coefficient matrix as follows (see [19] for the details of its formal algorithm).

Algorithm 1 (Extraction of coefficient matrix)**.** Let $j, k \in \{1, \ldots, m\}$. Let $\begin{pmatrix} x_1 & \cdots & x_m \end{pmatrix}^T \leftarrow \begin{pmatrix} e_1 & \cdots & e_m \end{pmatrix}^T$ be a given vectorized assignment. Here, we regard e_j as a function over a given semiring whose parameters are $x_1, \ldots, x_m$. An expression e' denotes an entry of an extracted coefficient matrix; e.g., e'_{jk} is the (j, k)-th entry. A coefficient matrix is extracted as follows:

- e'_{jk} is obtained by differentiating e_j with respect to x_k. Then, the linearity of e_j is also checked.
- $e'_{j(m+1)}$ is obtained by substituting $\mathbf{0}$ to all accumulators in e_j.
- $e'_{(m+1)k} = \mathbf{0}$, and $e'_{(m+1)(m+1)} = \mathbf{1}$.

After these two phases, the normal form is immediately obtained. We summarize the overall algorithm as follows.

Algorithm 2 (Extraction of normal form)**.** Let b be a loop body. Semirings are given.

1. Split b into b_1 and b_2, where b_1 contains only updating of each accumulator and b_2 corresponds to a residual computation, Note that b_1 is a sequence of assignments.
2. Convert b_1 into a vectorized assignment.
3. Extract a coefficient matrix from the right side of b_1 by using Algorithm 1; Try it for all semirings until it succeeds.
4. Transform b_1, which is in matrix multiplication form, into a normal form and insert b_2 into its auxiliary part.

Due to space limitations, we omit the algorithm used to transform a loop body into a sequence of separable normal forms. The main difference from Algorithm 2 is to analyze the dependence among accumulators. See [19] for the details.

4.2 Heuristic for Division

When an accumulator in a loop body occurs in an expression of divisors or dividends, the body cannot be transformed into a normal form by using only the axioms of a semiring. However, the axioms of another algebra can be used to transform such a loop body into a normal form. For example, consider the following loop:

for $i = 1$ **to** n **do**
$x \leftarrow (a[i] \cdot x + b[i])/(c[i] \cdot x + d[i]); \ y[i] \leftarrow x$ **done**.

In this body, a Möbius transformation is recurrently applied to x. Let f_i be this Möbius transformation. By means of the projective matrix representation of f_i, we obtain

$$(f_n \circ \cdots \circ f_1)(z) = (d \circ f'_n \circ \cdots \circ f'_1) \left(\begin{pmatrix} z & 1 \end{pmatrix}^T \right),$$

$$\textbf{where } f_i(z) = \frac{a[i] \cdot z + b[i]}{c[i] \cdot z + d[i]}, \ f'_i(\boldsymbol{z}) = \mathfrak{H}_i \times \boldsymbol{z},$$

$$\mathfrak{H}_i = \begin{pmatrix} a[i] & b[i] \\ c[i] & d[i] \end{pmatrix}, \ d\left(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \right) = \frac{w_1}{w_2}.$$

By using this relation, we can transform the above loop into the following loop with a body in normal form:

$x_1 \leftarrow x; \ x_2 \leftarrow 1;$
for $i = 1$ **to** n **do**
$\boldsymbol{t} \leftarrow A \times_{\{+,\cdot\}} \boldsymbol{x}; \ y[i] \leftarrow t_1/t_2; \ \boldsymbol{x} \leftarrow \boldsymbol{t}$ **done**,

$$\textbf{where } \boldsymbol{t} = \begin{pmatrix} t_1 \\ t_2 \\ 1 \end{pmatrix}, \ A = \begin{pmatrix} a[i] & b[i] & 0 \\ c[i] & d[i] & 0 \\ 0 & 0 & 1 \end{pmatrix}, \ \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}.$$

We can generalize this transformation over a division semiring[2]. $(\mathbb{R}, +, \cdot)$ and $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$ are useful instances of division semirings. Note that the division over $(\mathbb{R}, +, \cdot)$ is $/$, that over $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$ is $-$.

Theorem 3. *A recurrence equation that applies a Möbius transformation over a division semiring to an accumulator can be transformed into a normal form.*

We use Theorem 3 as a heuristic for divisions in extracting normal forms. We omit the details of its transformation algorithm due to space limitations. See [19] for the details.

5. Max-operator Extraction

The max-operator $\uparrow$ is not a built-in operator in most languages, but, as shown in the parallelization of Example 2, it is quite helpful for parallelization. In this section, we describe a technique for extracting max-operators from if-statements by exploiting an SMT solver, which is, informally, a SAT solver extended to handle numbers, arrays, conditional expressions, etc.

First, we convert a loop body into a conditional vectorized assignment (CVA), whose right side is a conditional expression

[2] A division semiring is similar to a semiring, but with requirement that each element must have a multiplicative inverse.

returning a vector. For example, consider the following loop body[3]:

if $x_1 + a[i] > 0$ **then** $x_1 \leftarrow x_1 + a[i]$ **else** $x_1 \leftarrow 0$ **endif**;
if $x_1 > x_2$ **then** $x_2 \leftarrow x_1$ **else endif**.

By iterating symbolic substitution, we convert this body into an equivalent CVA:

$$\begin{aligned}\boldsymbol{x} \leftarrow (x_1 + a[i] > 0)\ ?\ (x_1 + a[i] > x_2)\ ?\ \boldsymbol{x}'_1 : \boldsymbol{x}'_2 \\ :\ (0 > x_2)\ ?\ \boldsymbol{x}'_3 : \boldsymbol{x}'_4,\end{aligned}$$

$$\textbf{where } \boldsymbol{x}'_1 = \begin{pmatrix} x_1 + a[i] \\ x_1 + a[i] \end{pmatrix},\ \boldsymbol{x}'_2 = \begin{pmatrix} x_1 + a[i] \\ x_2 \end{pmatrix},$$

$$\boldsymbol{x}'_3 = \begin{pmatrix} 0 \\ 0 \end{pmatrix},\ \boldsymbol{x}'_4 = \begin{pmatrix} 0 \\ x_2 \end{pmatrix},\ \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

Next, we eliminate infeasible subexpressions from the right side of the CVA. We can use an SMT solver to detect them. In the above CVA, the right side has no infeasible subexpression. Now observe that $\boldsymbol{x}'_1$, $\boldsymbol{x}'_2$, $\boldsymbol{x}'_3$, and $\boldsymbol{x}'_4$ can be assigned to $\boldsymbol{x}$. If it is possible to convert the conditional expressions into max-expressions, we obtain a vectorized assignment:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} (x_1 + a[i]) \uparrow 0 \\ (x_1 + a[i]) \uparrow x_2 \uparrow 0 \end{pmatrix}. \tag{1}$$

This is only an assumption. Then, we confirm this assumption by checking that each assigned value is larger than or equal to other candidate values under the precondition for assigning the value. In the above example, for $x_1 \leftarrow (x_1 + a[i]) \uparrow 0$, we check

$$\begin{aligned}&\forall x_1, x_2, a[i](x_1 + a[i] > 0 \wedge x_1 + a[i] > x_2 \rightarrow x_1 + a[i] \geq 0),\\ &\forall x_1, x_2, a[i](x_1 + a[i] > 0 \wedge x_1 + a[i] \leq x_2 \rightarrow x_1 + a[i] \geq 0),\\ &\forall x_1, x_2, a[i](x_1 + a[i] \leq 0 \wedge x_1 + a[i] > x_2 \rightarrow 0 \geq x_1 + a[i]),\\ &\forall x_1, x_2, a[i](x_1 + a[i] \leq 0 \wedge x_1 + a[i] \leq x_2 \rightarrow 0 \geq x_1 + a[i]).\end{aligned}$$

To encode this problem into satisfiability problems, we convert $\forall$ into $\exists$ by negating each formula. Then, by using an SMT solver, we test the satisfiability of

$$\begin{aligned}&x_1 + a[i] > 0 \wedge x_1 + a[i] > x_2 \wedge x_1 + a[i] < 0,\\ &x_1 + a[i] > 0 \wedge x_1 + a[i] \leq x_2 \wedge x_1 + a[i] < 0,\\ &x_1 + a[i] \leq 0 \wedge x_1 + a[i] > x_2 \wedge 0 < x_1 + a[i],\\ &x_1 + a[i] \leq 0 \wedge x_1 + a[i] \leq x_2 \wedge 0 \geq x_1 + a[i].\end{aligned}$$

If all these formulae are unsatisfiable, we obtain $x_1 \leftarrow (x_1 + a[i]) \uparrow 0$. We similarly obtain $x_2 \leftarrow (x_1 + a[i]) \uparrow x_2 \uparrow 0$. Finally, we obtain (1), a set of linear recurrence equations over $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$, which we can parallelize on the basis of Theorem 2.

The steps above are the key to extracting max-operators. If they are applied naïvely, however, max-operators may not be extracted from a CVA that has a symbolically constant condition-part on the right side. For example, consider the loop body

if $x + a[i] \geq 0$ **then** $x \leftarrow (a[i] > 0)\ ?\ x + a[i] : x + a[i] + 1$
else $x \leftarrow 0$ **endif**.

One set of candidate values assigned to x is $\{x + a[i], x + a[i] + 1, 0\}$, and the assumption from this set is $x \leftarrow (x + a[i]) \uparrow (x + a[i] + 1) \uparrow 0$. Because $x + a[i] < x + a[i] + 1$, this assumption is incorrect. However, another set is $\{(a[i] > 0)\ ?\ x + a[i] : x + a[i] + 1, 0\}$, and the assumption from this set is $x \leftarrow ((a[i] > 0)\ ?\ x + a[i] : x + a[i] + 1) \uparrow 0$. This assumption is correct. A conditional expression whose condition-part is a symbolically constant expression is *unharmful* for parallelization. Therefore, before extraction of max-operators, we make an unharmful conditional expression coalesce into a vector. For example, we make the following conditional expression:

$$c\ ?\ \begin{pmatrix} e_{11} \\ e_{12} \end{pmatrix} : \begin{pmatrix} e_{21} \\ e_{22} \end{pmatrix}, \quad \textbf{where } c \in \mathcal{C},$$

coalesce into the following vector:

$$\begin{pmatrix} c\ ?\ e_{11} : e_{21} \\ c\ ?\ e_{21} : e_{22} \end{pmatrix}.$$

We summarize these techniques as follows.

Algorithm 3 (Extraction of max-operators)**.** A CVA whose right side, e, has no infeasible subexpression is given. Iterate Steps 1–4 until e becomes a vector.

1. Make all unharmful conditional expressions in e coalesce into vectors.
2. Extract the most deeply nested conditional subexpression whose all condition-parts are recurring expressions from e.
3. Test the extracted one on the basis of its precondition (using an SMT solver) and obtain its equivalent max-expression.
4. Replace the extracted conditional expression in e with its equivalent max-expression.

We can simply incorporate Algorithm 3 into Algorithm 2. After Step 1, we convert a reduction part (b_1) into a singly nested if-statement, and then convert it into an equivalent CVA, by inserting self-assignments as needed. Next, we eliminate its infeasible subexpressions (possibly by using an SMT solver). Then, we apply Algorithm 3 to this CVA and obtain a vectorized assignment. The rest of the process follows Steps 3 and 4 of Algorithm 2.

6. Optimizations for Our Framework

Unfortunately, our parallelization imposes intrinsic overhead on parallelized programs. Consider a sequential loop whose body is in matrix multiplication form and its parallelized version. The original sequential version iterates matrix-vector multiplication, while the parallelized version iterates matrix-matrix multiplication. Hence, overhead in proportion to the size of a coefficient matrix is imposed on the parallelized version. To obtain good performance in practice, optimizations to minimize intrinsic overhead are important. In this section, we describe several optimizations for parallelized programs. Due to space limitations, we do not explain all optimizations. See [19] for the details.

6.1 Accurate Method of Abstract Matrix Multiplication

Matsuzaki et al. [13] presented an optimization on the basis of abstract matrix multiplication[4] for parallelized reductions on trees. It eliminates redundancy in matrix multiplication. Their original method is a *conservative* one. We have developed an *accurate* method for loops parallelized by our framework.

We describe how to optimize a loop whose body is a matrix multiplication form with a coefficient matrix A. First, we abstract the values of the entries of A to four values Z, I, C, and V. Z and I denote $\mathbf{0}$ and $\mathbf{1}$, respectively. C denotes any constant value other than $\mathbf{0}$ and $\mathbf{1}$. V denotes any non-constant value, i.e., the value of a variable. Then, we simulate runtime matrix multiplication over abstract matrices. Let A^*_0 be the abstract matrix of A. Updating of abstract matrices is defined as $A^*_i = A^*_0 \times_{\{\oplus^*, \otimes^*\}} A^*_{i-1}$, where $i \geq 1$ and the semantics of $\oplus^*$ and $\otimes^*$ is defined in Figure 1. Here, an updating series of abstract matrices (i.e., $A^*_0, \ldots, A^*_i, \ldots$) reaches a stationary state (constant or periodic) since the size of each matrix is fixed and the domain of each entry is finite.

[3] A loop that has this body is part of mss described in Section 7.

[4] Note that [13] contains incorrect descriptions concerning abstract matrix multiplication, which were fixed in [11].

$\oplus^*$	Z	I	C	V
Z	Z	I	C	V
I	I	V	V	V
C	C	V	V	V
V	V	V	V	V

$\otimes^*$	Z	I	C	V
Z	Z	Z	Z	Z
I	Z	I	C	V
C	Z	C	V	V
V	Z	V	V	V

Figure 1. Semantics of two operators over the four abstract values.

Intuitively, A_i^* indicates how many variables we require for accumulating exponentiation of A and how we update these variables with compile-time constants. For example, the updating series of abstract matrices defined as follows converges at a constant state:

$$A_0^* = \begin{pmatrix} I & V \\ Z & I \end{pmatrix}, \quad A_i^* = \begin{pmatrix} V & V \\ Z & I \end{pmatrix} \quad (i \geq 1).$$

This indicates that, once we have computed a square of A, we require always only two loop-carried scalar variables for exponentiating A. Concretely, A^n over $(R, \oplus, \otimes)$ can be implemented as follows:

$$i \leftarrow 0;\ A' \leftarrow A;\ i \leftarrow 1;\ A'' \leftarrow A \times_{\{\oplus,\otimes\}} A';$$
$$\textbf{for } i = 2 \textbf{ to } n \textbf{ do } A'' \leftarrow A \times_{\{\oplus,\otimes\}} A'' \textbf{ done},$$
$$\textbf{where } A' = \begin{pmatrix} \mathbf{1} & v_2 \\ \mathbf{0} & \mathbf{1} \end{pmatrix}, \quad A'' = \begin{pmatrix} v_1 & v_2 \\ \mathbf{0} & \mathbf{1} \end{pmatrix}.$$

For convergence at a finite set of periodic states, we convert it into a constant state by loop unwinding. Due to space limitations, we omit the details. See [19] for the details.

The primary effect of this technique is, of course, elimination of non-trivial redundancy; i.e., it realizes non-trivial copy propagation and constant folding. In addition, this technique has an important secondary effect: elimination of storing $\mathbf{0}$ into variables. $\mathbf{0}$ over $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$ is $-\infty$. This is a troublesome value because its annihilation property (i.e., $-\infty + a = a + (-\infty) = -\infty$) is difficult to efficiently implement. This technique enables us to identify and eliminate $\mathbf{0}$ in compile-time.

The main difference between our method and Matsuzaki et al.'s [13] is the semantics of updating of abstract matrices. Their method is conservative with respect to convergence, i.e., over-approximates the stationary state of an updating series of abstract matrices. As a result, their method cannot deal with periodic states effectively and avoid storing $\mathbf{0}$ into variables, whereas our method can do these successfully by using loop unwinding together.

6.2 Splitting up shift

Sometimes, reduce and scan computations include a kind of doall computation. Such computation appears in a coefficient matrix as its row whose entries are $\mathbf{0}$ except for the rightmost entry. For example, consider the following loop:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } \boldsymbol{x} \leftarrow A \times_{\{+,\cdot\}} \boldsymbol{x} \textbf{ done},$$
$$\textbf{where } A = \begin{pmatrix} a[i] & 1 & 0 \\ 0 & 0 & a[i] \\ 0 & 0 & 1 \end{pmatrix},\ \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}.$$

The second row, i.e., $x_2 \leftarrow a[i]$, represents doall computation. We can split up the row by introducing a temporary array t as follows:

$$\textbf{for } i = 1 \textbf{ to } n-1 \textbf{ do } t[i] \leftarrow a[i] \textbf{ done};\ t[0] \leftarrow x_2;$$
$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } \boldsymbol{x}' \leftarrow A' \times_{\{+,\cdot\}} \boldsymbol{x}' \textbf{ done},$$
$$\textbf{where } A' = \begin{pmatrix} a[i] & t[i-1] \\ 0 & 1 \end{pmatrix},\ \boldsymbol{x}' = \begin{pmatrix} x_1 \\ 1 \end{pmatrix}.$$

The first loop is obviously a doall loop. The second loop, whose body is in the matrix multiplication form, is more efficient than the original loop because the size of A' is smaller than that of A. This efficiency make an effect on matrix-matrix multiplication of the parallelized version. Although this transformation causes $\mathrm{O}(n)$ space overhead due to the introduction of t, this space overhead is completely eliminated by *fusion* as mentioned later.

This split up doall computation corresponds to shift [7], a pattern of parallel computation, and this transformation corresponds to the combination of scalar expansion and loop fission in doall parallelization.

6.3 Fusion

Consider a scan followed by a reduce. We can compute both the local scan of the preceding scan and the local reduction of the succeeding reduce simultaneously. If the result of the preceding scan is used only by the succeeding reduce, we can eliminate $\mathrm{O}(n)$ space for the intermediate data that the scan produces and that the reduce consumes. Such kind of transformation is called *fusion*. The transformation described above is the fusion of scan and reduce [12]. It is, of course, applicable to ones extracted by our framework. Furthermore, the fusion of shift with scan or reduce [7] is also applicable. We omit the details of these fusion methods.

6.4 Specialization of Final Value Guarantee

Sometimes, a loop body contains the following assignment:

$$x \leftarrow c_1\ ?\ x : c_2, \quad \textbf{where } c_1, c_2 \in \mathcal{C}.$$

We observe that this updating of x does not use the current value of x because self-assignment means no updating. Based on this observation, we can compute the result of x in parallel by using a parallel loop and a reduce. For example, consider the following loop derived from parallelization process of Example 4:

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } x_2 \leftarrow (t[i] + a[i] \geq 0)\ ?\ x_2 : i + 1 \textbf{ done}.$$

Its parallelized version is:

$$\textbf{parallel } k \in \{1, \ldots, p\} \textbf{ do}$$
$$v[k] \leftarrow x_2;\ b[k] \leftarrow \text{False};$$
$$\textbf{for } i = 1 + (k-1)n/p \textbf{ to } kn/p \textbf{ do if } t[i] + a[i] \geq 0 \textbf{ then}$$
$$\textbf{else } v[k] \leftarrow i + 1;\ b[k] \leftarrow \text{True } \textbf{endif done}$$
$$\textbf{done};\ \boldsymbol{x} \leftarrow \mathsf{reduce}(\triangleright, [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p]),$$
$$\textbf{where } \boldsymbol{x} = \begin{pmatrix} x_2 \\ b[0] \end{pmatrix},\ \boldsymbol{v}_i = \begin{pmatrix} v[i] \\ b[i] \end{pmatrix},\ \boldsymbol{v}_i \triangleright \boldsymbol{v}_j = (b[j])\ ?\ \boldsymbol{v}_j : \boldsymbol{v}_i.$$

In this program, v is an array whose k-th element is the partial result computed by the k-th thread and b is an array whose k-th element is the flag to denote that the k-th thread update its partial result. Because this parallelized program does not use matrix multiplication, it is more efficient than one shown in Section 3.3.

From the viewpoint of doall parallelization, this computation corresponds to a final value guarantee for a conditional assignment. In doall parallelization, this kind of guarantee has not been dealt seriously with. In fact, ICC 11.1 with `-parallel -par-threshold0` options did not parallelize a loop that contains this assignment because ICC cannot deal with it. Therefore, the presented technique here is also helpful for doall parallelization.

7. Experiments

We have implemented our framework and techniques as a parallelizer in a compiler infrastructure, COINS[5], which includes a C frontend, a doall analyzer, and a C code generator. Our implementation employs an SMT solver, Yices[6]. Figure 2 shows the entire

[5] `http://coins-compiler.sourceforge.jp/international/`

[6] `http://yices.csl.sri.com/`

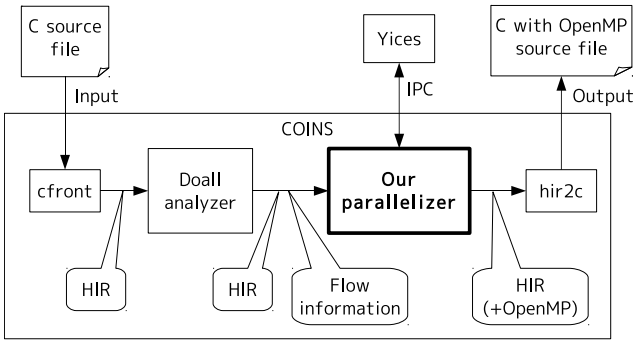

Figure 2. The entire structure of our implementation. HIR is the high-level intermediate representation of COINS. Our parallelizer communicates with Yices through inter-process communication.

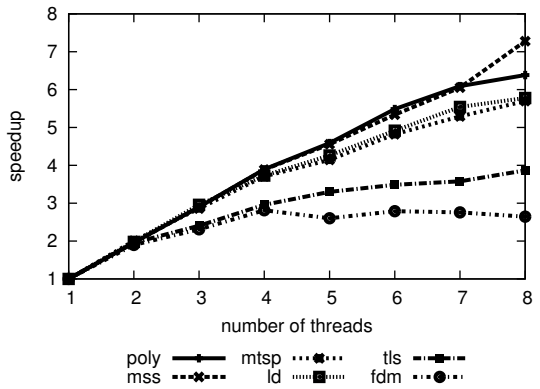

Figure 3. Scalability of parallelized versions of six examples.

structure of our implementation. Our implementation produces parallelized C programs that employ OpenMP constructs.

To confirm the scalability of programs parallelized by our implementation, we conducted experiments on the following example programs, each of whose doall parallelization fails.

poly It computes polynomial evaluation through the Horner scheme. Its parallelized version executes reduce with $\times_{\{+,\cdot\}}$ of 2-by-2 matrices. The order of a given polynomial was 2^{27} and it was evaluated with `double` arithmetic.

mss It solves the maximum segment sum problem, which is to compute the maximum sum of a contiguous segment of a given array. It is known as a programming pearl. Its parallelized version executes reduce with $\times_{\{\uparrow,+\}}$ of 3-by-3 matrices. The length of a given array was 2^{27} and its each element was `int`.

mtsp It is equivalent to Example 4; it computes the mts of a given array and its start position. Its parallelized version executes scan with $\times_{\{\uparrow,+\}}$ of 2-by-2 matrices and executes the final value guarantee for a conditional assignment. The length of a given array was 2^{27} and its each element was `int`.

ld It computes the Levenshtein distance between two given strings through dynamic programming. Our parallelization worked to divide the longer sides of the memo table whose element was `int`. Its parallelized version executes scan with $\times_{\{\uparrow,+\}}$ of 2-by-2 matrices. The length of one given string of `char` was 2^{27} and that of the other was 6.

tls It is a solver for tridiagonal linear systems. It consists of three parallelizable loops. The first executes LU decomposition, i.e., obtains $LU\boldsymbol{x} = \boldsymbol{b}$. It was parallelized by using Theorem 3. Its parallelized version executes scan $\times_{\{+,\cdot\}}$ of 3-by-3 matrices. The second solves $L\boldsymbol{y} = \boldsymbol{b}$, and the third solves $U\boldsymbol{x} = \boldsymbol{y}$. The parallelized versions of the second and third execute scan with $\times_{\{+,\cdot\}}$ of 2-by-2 matrices. The parallelized solver is equivalent to [20]. The dimension of a given system was 2^{26} and it was solved with `double` arithmetic.

fdm It solves a one-dimensional heat equation through an *in-place* finite difference method (FDM). The parallelization of standard implementations of FDM to use a buffer where values in the next time-step are stored is trivial, but that of in-place ones is non-trivial. Its parallelized version does not compute matrix multiplication because its coefficient matrix can shrink completely through splitting up shift and its fusion. Its parallel computation is thus done only through the local reduction. The number of space cells was 2^{27}, that of time steps was 10, and the values of temperature were `double`.

	poly	mss	mtsp	ld	tls	fdm
seq. (ms)	360	522	351	1807	487	2483
par. (ms)	578	265	633	2383	1083	3261
seq. / par. (%)	62.2	197	55.4	75.8	45.1	76.1

Table 1. Execution time of original sequential versions and parallelized single-threaded versions of six examples.

p	1	2	3	4	5	6	7	8
r	1.73	1.80	2.41	2.22	2.56	2.63	3.02	3.44

Table 2. Ratio of speed of our parallelized mss to that of Fisher and Ghuloum [8]'s mss; r and p denote ratio of speed (higher is better) and number of threads, respectively.

We used a server equipped with dual Xeon X5550 (4 cores; 3.06 GHz; HT disabled) and 12 GB of memory (DDR3-1333) running Ubuntu 10.04 (64-bit), and we compiled each program to its executable by employing ICC 11.1 with the `O3` optimization.

As shown in Figure 3, the parallelized versions of poly, mtsp, mss, and ld achieved near-linear scalability and nearly 6 or more times speeds with 8 threads. These results reflected the theoretical scalability (shown in Theorem 2). In contrast, the speed of parallelized fdm peaked out on 4 threads and the speedup of parallelized tls degraded. Because these do relatively much memory access, memory bandwidth bound their performance. As mentioned in Section 6, our parallelization causes intrinsic runtime overhead. Table 1 shows the execution time of the original sequential version and parallelized single-threaded version for each program. Each example had expected performance, except for mss. This is because the original sequential version of mss was abnormally slow for unknown reasons. However, this aberration is independent of the correctness and utility of our parallelization. Overall, without memory bandwidth issues, our parallelization achieved good scalability.

Fisher and Ghuloum [8]'s method parallelizes mss automatically into the composition of a scan and reduction, unlike ours. We tested the version parallelized by using their method. Table 2 shows the ratio of the speed of our parallelized mss to that of their parallelized mss. Our framework and techniques enabled the compiler to produce the more efficient parallel mss than theirs.

The SMT problem is not easier than the SAT problem, which is NP-complete. Therefore, to exploit an SMT solver requires high costs potentially. Our implementation exploits Yices to extract max-operators. Our implementation can dump each query for Yices as a file. We measured the execution time of Yices for queries.

example	mss	mtsp	ld
number of queries	8	2	2
total time (ms)	11	4	4

Table 3. Execution time for Yices to extract max-operators.

Yices's version was 1.0.27. Table 3 shows the results. The execution time for each program was short enough for practical use. Moreover, because we can test each query independently, we can obtain better performance by exploiting process-level parallelism. Therefore, our approach based on an SMT solver is very practical.

8. Discussion

8.1 Prior Work and Our Improvement

The most significant prior work is one by Fisher and Ghuloum [8]. They pioneered in automatic parallelization of loops by extracting associative operators. We briefly describe their formalization and technique, and compare ours with theirs.

Their key observation is that, owing to the associativity of the function composition, application of a composite function can compute in parallel by computing the function composition per se. Their key idea is that, if a function is closed under composition, its composition can be computed efficiently. For example, $f_i(x) = a_i x + b_i$, f_i is closed under composition: $(f_1 \circ f_2)(x) = (a_1 a_2)x + (a_1 b_2 + b_1)$. The computation of $\circ$ in $f_1 \circ f_1$ corresponds to that of $a_1 a_2$ and $a_1 b_2 + b_1$. Concretely, they formalized a loop that computes reduction or scan as an application of a composite function consisting of its *modeling function*, which represents its loop body, and then parallelized the loop into a version that computes the composition of the modeling function in parallel.

Obviously, their formalization is more general and powerful than ours. The modeling function of the matrix multiplication form is the affine function over a semiring, which is closed under composition. Hence, our formalization is only a special case of theirs. However, our framework is *more practical* because it is easier to implement in compilers and to derive efficient parallel programs.

The test of closure under composition is potentially very difficult and costly because it necessitates searching in enormous space. This difficulty derives from a fact that proof of disclosure by counterexample is almost impossible because the closure property is too abstract. In contrast, to derive matrix multiplication, we have only to check the linearity of recurrence equations. The linearity is easier to check, and moreover we can find a counterexample of the linearity immediately.

Of course, they had noticed this difficulty. They developed a heuristic to test the closure by checking structural isomorphism of simplified nested conditional expressions. However, this approximation has several problems. First, the generality of this heuristic is such a little that it managed to enable us to extract a single max-operator. Second, to avoid a complicated conditional expression, they split a loop body into as small ones as possible. Then, they parallelized computations for dependent accumulators as independently as possible. As a result, they transform a loop potentially parallelized as a single reduction into a composition of a reduction and scans. In fact, they parallelized mss as a composition of a scan and reduction, whose performance is worse than its single-reduction version (see Table 2).

In contrast, we extract max-operators before trying parallelization. We can then easily parallelize recurrence equations over $(\mathbb{R} \cup \{-\infty\}, \uparrow, +)$. Matrix multiplication can deal with dependent (as well as independent) accumulators as a vector. Therefore, our framework can derive a single reduction from computations for dependent accumulators, e.g., mss. In addition, they did not perform exhaustive search for its complexity, but we easily do it by exploiting an SMT solver. Focusing on max-operators significantly simplifies analysis for parallelization and affords optimizations.

Overall, our framework focuses on techniques rather than formalism and on practicality rather than generality. In spite of the less generality, our framework can deal with all their examples, and furthermore, derive more efficient programs than theirs did. It demonstrates that the power of our framework is empirically no less than that of theirs, and that our framework is more practical.

8.2 Parallelization of Recursive Functions

Matsuzaki et al. [13] dealt with parallelization of a reduction of trees and presented a model of matrix multiplication over a semiring. Hence, our formalization per se is not so novel and is a variation specialized for loops. Their framework does not deal with accumulation and assumes the max-operator to be given. Although their semi-automated code generator is equipped with an optimization mechanism (see also Section 6.1), they did not show any experimental result on the performance of parallelized programs. Therefore, we do not consider their work as fully automatic parallelization in practical situations.

Chin et al. [6] presented the *context preservation* theorem that formalizes the idea of [8] in a functional language, and an algorithm for parallelizing a linear self-recursive function into a list homomorphism [4], which is a naturally parallelizable recursive form.

Xu et al. [21] presented a type-based approach to parallelization on the basis of the context preservation theorem. Their focus was on the analysis that uses an algebra similar to a semiring even though their implementation can generate list homomorphisms. They hardly dealt with higher-order linear recurrences, whereas we have dealt seriously with them. Since they assumed the max-operator to be given, did not deal with optimization, and conducted no experiment on programs parallelized by their implementation, we therefore judge that their work is not one for fully automatic parallelization in practical situations.

Morihata and Matsuzaki [14] presented that quantifier elimination by virtual substitution (QEVS) enables us to directly extract associative operators based on the context preservation theorem from recursive functions, and demonstrated that their QEVS-based implementation with simple heuristics can parallelize non-linear, non-self-recursive, and accumulative functions. Although QEVS is modular, however, QEVS is too costly to implement in compilers, and their technique has less scalability to the number of variables.

Morita et al. [15] presented an algorithm to extract associative operators based on inversion of functions and the third homomorphism theorem. Their work is very interesting, but their parallelization is impractical because the third homomorphism theorem necessitates two equivalent functions for one target problem.

8.3 Linear Recurrence Equation and Scan

Apart from work on automatic parallelization, Kogge and Stone [10] presented a parallel algorithm, which is now an algorithm of scan, for solving a general class of linear recurrence equations by formalizing the class with matrix operations. Stone [20] presented a parallel algorithm for solving a tridiagonal linear system on the basis of scan and a projective matrix of Möbius transformation. Cyclic reduction [9] is another parallel algorithm for solving it.

From this historical background, it is quite natural to parallelize computation of linear recurrence equations through scan with matrix multiplication. Although generalization over an algebra (e.g., a semiring) is not explicitly mentioned in [10, 20], these algorithms are sufficiently generic. Therefore, there is no difficulty in dealing with computation of linear recurrence equations. From the perspective of automatic parallelization, the difficulty is to discover this computation from a given loop *automatically*.

Redon and Feaurier [17] presented methods to detect *sequential* scans in the polytope model. With respect to parallelization, their methods fundamentally depend on the detection of trivial built-in associative operators by pattern matching. Although they mentioned derivation of matrix multiplication from a first-order linear recurrence equation and Möbius transformation, they considered each of these as a special case and did not generalize these at all. They did not deal with parallelization of a loop that has a complicated body such as one containing if-statements and one containing computations other than linear recurrence equations.

Nistor et al. [16] presented optimization techniques for matrix multiplication derived from parallelized linear recurrence equations. Their techniques reduce the space used in matrix multiplication by recovering an expression of one entry from an expression of another entry on the basis of linear relations between these expressions. Because their techniques are independent of ours, we can apply them to our framework.

8.4 Standard Approach in Doall Parallelization

Doall parallelization (and also vectorization [3]) fundamentally necessitates eliminating every loop-carried data dependence in target loops by employing loop transformation. For example, loop skewing [2] eliminates loop-carried data dependence and exposes wavefront parallelism in a nested loop such as ld. Whereas our framework incorporates reduction as a fundamental part, doall parallelization handles reduction as a special case of eliminating loop-carried data dependence.

The book on the standard doall parallelization [2] tells reductions have three essential properties: 1) *"they reduce the elements of some vectors or array dimension down to one element"*, 2) *"only the final result of the reduction is used later; use of an intermediate result voids the reduction"*, and 3) *"there is no variation inside the intermediate accumulation; that is, the reduction operates on the vector and nothing else."* Besides, reduction operators are almost always assumed as built-in associative commutative ones.

Our framework relaxes these properties that are regarded as essential ones. Because our framework can parallelize computation for mutually dependent accumulators, our framework relaxes the first. The use of `scan` overcomes the second and third. Therefore, our framework is a natural generalization of doall parallelization.

9. Conclusion

We have presented novel techniques to parallelize various loops. Our techniques extend doall parallelization.

We think that there is room for future work regarding dealing with nested loops. The most popular framework for optimizing nested loops is the polyhedral model, which often accompanies parallelization. The tie-up between our techniques and the polyhedral model is an innovative work. In addition, we think that our technique to extract max-operators is helpful for vectorization. Therefore, the exploitation of fine-grained data parallelism by using our techniques is another innovative work.

Acknowledgments

We would like to thank Masato Takeichi and Zhenjiang Hu for encouraging our research. We are grateful to Akimasa Morihata, Kento Emoto, and Kiminori Matsuzaki for technical discussions with the first author.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, 2006.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2001.

[3] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic Intra-Register Vectorization for the Intel®Architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002.

[4] R. S. Bird. An Introduction to the Theory of Lists. In *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, 1987.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*, pages 177–187. ACM, 1990.

[6] W.-N. Chin, A. Takano, and Z. Hu. Parallelization via Context Preservation. In *Proceedings of IEEE International Conference on Computer Languages (ICCL '98)*, pages 153–162. IEEE CS Press, 1998.

[7] K. Emoto, K. Matsuzaki, Z. Hu, and M. Takeichi. Domain-Specific Optimization Strategy for Skeleton Programs. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 705–714. Springer, 2007.

[8] A. L. Fisher and A. M. Ghuloum. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 135–146. ACM, 1994.

[9] W. Gander and G. H. Golub. Cyclic Reduction – History and Applications. In *Proceedings of the Workshop on Scientific Computing*, 1997.

[10] P. M. Kogge and H. S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Trans. Comput.*, 22(8):786–793, 1973.

[11] K. Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, 2007.

[12] K. Matsuzaki and K. Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *Implementation and Application of Functional Languages (IFL '09)*, volume 6041 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2010.

[13] K. Matsuzaki, Z. Hu, and M. Takeichi. Towards Automatic Parallelization of Tree Reductions in Dynamic Programming. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06)*, pages 39–48. ACM, 2006.

[14] A. Morihata and K. Matsuzaki. Automatic Parallelization of Recursive Functions using Quantifier Elimination. In *Functional and Logic Programming (FLOPS '10)*, volume 6009 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2010.

[15] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 146–155, 2007.

[16] A. Nistor, W.-N. Chin, T.-S. Tan, and N. Tapus. Optimizing the parallel computation of linear recurrences using compact matrix representations. *J. Parallel Distrib. Comput.*, 69(4):373–381, 2009.

[17] X. Redon and P. Feautrier. Detection of Scans in the Polytope Model. *Parallel Algorithms Appl.*, 15(3–4):229–263, 2000.

[18] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Pub, 1993.

[19] S. Sato. Automatic Parallelization via Matrix Multiplication. Master's thesis, The University of Electro-Communications, 2011.

[20] H. S. Stone. An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *J. ACM*, 20(1):27–38, 1973.

[21] D. N. Xu, S.-C. Khoo, and Z. Hu. PType System: A Featherweight Parallelizability Detector. In *Programming Languages and Systems (APLAS '04)*, volume 3302 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2004.